# Semi-Automatic Restructuring of Offloadable Tasks for Many-Core Accelerators

Nishkam Ravi
NEC Labs America
Princeton, NJ, USA
nravi@nec-labs.com

Yi Yang
NEC Labs America
Princeton, NJ, USA
yyang@nec-labs.com

Tao Bao
Computer Science
Purdue University
tbao@cs.purdue.edu

Srimat Chakradhar
NEC Labs America
Princeton, NJ, USA
chak@nec-labs.com

## ABSTRACT

Work division between the processor and accelerator is a common theme in modern heterogenous computing. Recent efforts (such as LEO and OpenAcc) provide directives that allow the developer to mark code regions in the original application from which *offloadable tasks* can be generated by the compiler. Auto-tuners and runtime schedulers work with the options (i.e., offloadable tasks) generated at compile time, which is limited by the directives specified by the developer. There is no provision for offload restructuring.

We propose a new directive to add *relaxed semantics* to directive-based languages. The compiler identifies and generates one or more offloadable tasks in the neighbourhood of the code region marked by the directive. Central to our contribution is the idea of *sub-offload* and *super-offload*. In sub-offload, only a part of the code region marked by the developer is offloaded to the accelerator, while the other part executes on the CPU in parallel. This is done by splitting the index range of the main parallel loop into two or more parts and declaring one of the subloops as the offloadable task. Support is added to handle reduction variables and critical sections across subloops. Sub-offload enables concurrent execution of a task on the CPU and accelerator. In super-offload, a code region larger than the one specified by the developer is declared as the offloadable task (e.g., a parent loop). Super-offload reduces data transfers between CPU and accelerator memory.

We develop *Elastic Offload Compiler*(EOC) for use alongside existing directive-based languages. The current implementation supports LEO for the new Intel Xeon Phi (MIC) architecture. We evaluate EOC with respect to SpecOMP and NAS Parallel Benchmarks. Speedups range between 1.3x-4.4x with the CPU version as baseline and 1.2x-24x with the offload (CPU-MIC) version as baseline.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*SC '13,* November 17-21 2013, Denver, USA.
Copyright is held by the author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503285

## 1. INTRODUCTION

Many-core coprocessors, such as GPUs, can provide order of magnitude better performance and efficiency for parallel workloads as compared to multicore CPUs, and are being widely adopted as accelerators for high performance computing. Nvidia GPUs, for example, are used in 13 of the top 100 supercomputers [13]. A recent addition to the line of many-core coprocessors is the x86-compatible Intel Xeon Phi (MIC) architecture, which is being incorporated in four of the top 100 supercomputers [13]. MIC is designed to leverage existing x86 experience and benefit from familiar parallelization tools, libraries and programming models, including OpenMP [9], CilkPlus [21], TBB [12], etc.

Accelerating parallel computation using many-core coprocessors requires specification of code regions that can be profitably offloaded to the coprocessor and executed as independent tasks. These code regions are typically created by the developer using low-level APIs (e.g., Cuda [2] and OpenCL [8]), which is known to be tedious and error-prone.

Recent efforts provide directives that allow the developer to mark code regions in the application from which offloadable tasks can be generated by the compiler. These efforts are intended to improve developer productivity and simplify code maintenance without requiring heroic static analysis.

LEO (Language Extensions for Offload) provides #pragma offload directive for marking offloadable code regions for Intel MIC and `in/out` clauses for orchestrating data transfers (as shown in Fig. 1). It is simple and general enough to be used with other coprocessor architectures and bears resemblance to contemporary efforts for GPUs: OpenAcc [7], OpenHMPP [1], OpenMPC [33] and OmpSs [6, 22] (as shown in Table 2). An in-depth comparison is provided in [35]. These languages provide support for asynchronous offload as well as optimizations for minimizing data transfers. Some of them (e.g., StarPU [15], OmpSs [6]) also support automatic task graph scheduling on heterogeneous processors.

Finding parts of the application that can be profitably executed on the accelerator (henceforth *offloadable tasks*) is left up to the developer. Once a code region has been specified by the developer as an offloadable task, there is no provision for task restructuring. This can result in increased development effort and sub-optimal performance, as discussed next.

### 1.1 Motivation

Legacy C/C++ applications are usually structured as one task with potentially multiple parallel loops embedded in the body of the application. These loops are the primary source of parallelism and need to be utilized fully for performance.

```
int g(..){
  ..
  for(i=0; i < K; i++){
    f();
    ..
  }
  #pragma omp parallel for
  for(j=0; j < M; j++){
    ..
  }
  ..
}

int f(..){
  ..
  /* hot loop */
  #pragma omp parallel for
  for(i=0; i < N; i++){
    A[i] = B[i] * C[i];
    ..
  }
  ..
}
```

(a)

```
int g(..){
  ..
  for(i=0; i < K; i++){
    f();
    ..
  }
  #pragma omp parallel for
  for(j=0; j < M; j++){
    ..
  }
  ..
}

int f(..){
  ..
  #pragma offload target(mic) in(B,C:
        length(N)) out(A:length(N))
  {
    /* hot loop */
    #pragma omp parallel for
    for(i=0; i < N; i++){
      A[i] = B[i] * C[i];
      ..
    }
  }
  ..
}
```

(b)

```
int g(..){
  ..
  for(i=0; i < K; i++){
    f();
    ..
  }
  #pragma omp parallel for
  for(j=0; j < M; j++){
    ..
  }
  ..
}

int f(..){
  ..
  #pragma offload target(mic) in(B,C:length(N))
       out(A:length(N)) signal(&flag)
  {
    /* hot loop */
    #pragma omp parallel for
    for(i=0; i < N; i++){
      A[i] = B[i] * C[i];
      ..
    }
  }
  .. // small code region R
  #pragma offload_wait target(mic) wait(&flag)
  {}
  ..
}
```

(c)

**Figure 1: Example: (a) CPU code (b) Heterogeneous (CPU-MIC) code (c) Advanced heterogeneous (CPU-MIC) code. None of these versions yields optimal performance.**

Fig. 1(a) shows a code snippet featuring a hot parallel loop in function `f()` being invoked from an outer serial loop in function `g()`. Let's assume that the parallel loop executes twice as fast on Intel MIC as compared to a multicore Xeon processor, and that the data transfer overheads are not high enough to offset the speedup. Offloading this parallel loop to MIC is expected to yield good performance gains. The developer would declare this loop as an offloadable task by inserting the offload directive, as shown in Fig. 1(b). CPU execution would suspend while the loop executes on the accelerator and resume upon completion; resulting in an overall gain in performance.

A savvy developer could go a step further and make the offload asynchronous by using `signal` and `wait` clauses, as shown in Fig. 1(c). In this case, CPU execution would continue past the pragma, after initiating the offload, execute small code region `R` and then block on `wait`. *The use of asynchronous offload primitives is effective only when the CPU has enough work to do (i.e., execute a large code region in parallel) while the offloaded task executes on the accelerator.* In the current example and in most typical applications, that is not the case.

We observe that: the parallel loop by itself is not the best candidate for offload, just the most obvious one; and that a better candidate would be the *subloop* obtained by splitting the index range of the parallel loop into two parts (as shown in Fig. 4(c)). The first subloop could be asynchronously offloaded and executed on the accelerator, while the second subloop executes on the CPU concurrently (i.e., constitutes code region `R`). This would minimize CPU idle time and improve performance, as confirmed by our experiments.

The second observation is that the serial loop in function `g()` can be a great candidate for offload, if certain conditions are met. By offloading the entire outer loop to the accelerator (as shown in Fig. 4(b)) data transfer overheads can be minimized. This is because the data would get transferred once for the entire loop, as opposed to once per iteration. Our experiments indicate that these savings can result in significant speedups. This is applicable, however, when the serial code contained inside the outer loop is relatively small, because serial performance of a many-core coprocessor is significantly lower than that of a multicore processor.

| | CPU Idle Time | Data Transfer Time |
|---|---|---|
| **330.art** | **62%** | 2% |
| **EP.C** | **89%** | 2% |
| **320.equake** | 5% | **64%** |
| **BT.B** | 4% | **71%** |

**Table 1: CPU idle time and data transfer time as fractions of total execution time when using Xeon Phi (MIC)**

Table 1 shows CPU idle time and data transfer time as fractions of the total execution time for four benchmarks from SpecOMP and NAS Parallel, when hot parallel loops are declared as offloadable tasks (using asynchronous offload primitives). In practice, optimal utilization of both CPU and accelerator is hard to achieve even with optimal task mapping and pipelining mechanisms, unless some amount of code restructuring is applied. Also, primitives that enable data reuse, such as `nocopy` in LEO and `present` in OpenAcc, do not fully exploit the opportunities for data reduction. Sub-offload and super-offload are designed to minimize processor idle time and data transfer overheads respectively, without relying on developer expertise.

## 1.2 Problem and Contribution

In the rest of the paper, we describe our contribution with respect to LEO and Intel MIC. The problem and solution, with suitable modification, can be applied to other directive-based dialects (and devices) as well.

Directives have *strict semantics*: #pragma offload indicates that the following code region must be treated as an offloadable task without fundamental modification. This puts the onus of identifying and specifying *optimal* offloadable tasks on the developer and limits offload restructuring. Auto-tuners and runtime schedulers have to pick among the options (i.e., offloadable tasks) generated at compile time, which in turn is limited by the directives specified by the developer. In their current form, directive-based languages do not allow offloadable tasks to be restructured and adapted by the compiler/runtime, they only allow tasks to be optimally mapped and scheduled, or advanced compiler opti-

Table 2: Key directives in common directive-based languages for accelerator programming

| | LEO | OpenAcc | OpenHMPP | OmpSs | OpenMPC |
|---|---|---|---|---|---|
| **offload** | `#pragma offload` | `#pragma acc kernels` | `#pragma hmpp codelet` | `#pragma omp task` | `#pragma cuda gpurun` |
| **synchronization** | `<signal,wait>` | `<async,wait>` | `<asynchronous,synchronize>` | `<input,output,taskwait>` | OpenMP `<nowait>` |
| **data transfer** | `<in,out,inout>` | `<copy,copyin,copyout>` | `args[item].io=<in,out,inout>` | `<copy_in,copy_out>` | `<c2gmemtr,g2cmemtr>` |
| **data reuse** | `nocopy` | `present` | `args[item].noupdate=true` | *by default* | `<nog2cmemtr,noc2gmemtr>` |

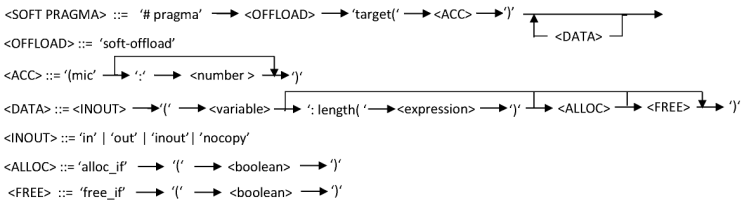



**Figure 2: Grammar for the new EOC directive**

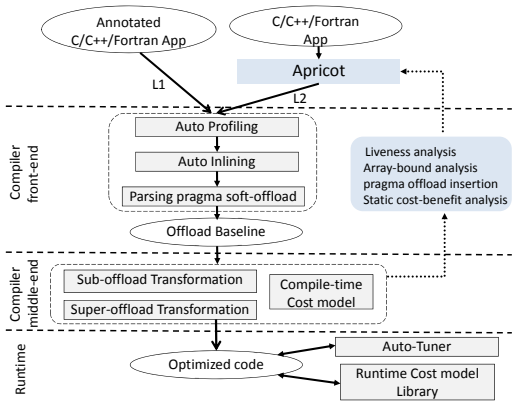



**Figure 3: EOC Overview (Apricot components invoked by EOC are shown in blue).**

mizations to be applied to the body of the task.

We propose a new directive called #pragma soft-offload to add *relaxed semantics* to directive-based languages and facilitate better collaboration between the developer and compiler. This directive allows the user to *suggest* rather than specify a parallel code region as an offloadable candidate. The compiler then identifies and generates one or more offloadable tasks for a code region marked by #pragma soft-offload. Source transformations are applied by the compiler as necessary. This directive is not intended as a replacement for the must-directive (#pragma offload).

Central to our contribution is the idea of *sub-offload* and *super-offload*. In sub-offload, only part of the code region marked by the developer is offloaded to the accelerator, while the other part executes on the CPU in parallel. This is done by splitting the index range of the main parallel loop into two or more parts and declaring one of the subloops as the offloadable task (henceforth *subtask*). The MIC loop shown in Fig. 4(c) is a subtask. In super-offload, a code region larger than the one specified by the developer is declared as the offloadable task (henceforth *supertask*). Supertask *contains* the code region marked by #pragma soft-offload. For the snippet in Fig. 1(b), supertask would be the outer serial loop in `g()`. Main contributions of this paper:

- We propose a new directive: #pragma soft-offload, to add relaxed semantics to directive-based languages
- Present compiler support for *sub-offload* and *super-offload* for enabling the new directive
- As part of sub-offload, we add support for handling critical regions and reduction variables in and across parallel loops without additional synchronization
- We present compile-time and runtime cost-models for selective application of these optimizations, based on the idea of *loop sampling*
- We evaluate our framework with respect to SpecOMP and NAS parallel benchmarks on Xeon Phi (MIC) coprocessor and show significant performance gains

The key features of our framework *Elastic Offload Compiler (EOC)* are as follows:

- It is intended as a performance and productivity tool to be used alongside existing directive-based languages, such as LEO and OpenAcc.
- As a source-to-source compiler, it is portable and does not require modification to the underlying runtime.
- Only one new directive is added to preserve *economy of mechanism* and minimize external complexity. The runtime library is small consisting of a few cost-model functions that return true or false.
- Sub-offload and super-offload can be applied as independent optimizations when offloadable code regions are identifiable without developer assistance.

## 2. EOC OVERVIEW

EOC is primarily a source-to-source compiler, which accepts an annotated C/C++/Fortran application as input and outputs optimized source code. It also has a small runtime library consisting of cost-model functions. Invocations to these cost-model functions are automatically inserted during code optimization. EOC is implemented on top of gcc [3] and Apricot [42]. The analyses are performed on gcc IR, but the transformations are applied to the source code, allowing the developer to make further modifications if needed. The resulting source code file(s) are fed into an auto-tuner and compiled with a C/C++/Fortran compiler, such as icc [5].

EOC can be invoked at two different optimizations levels: *L1* and *L2*. At L1, the incoming application is assumed to have been fully annotated with offload directives. EOC lowers #pragma soft-offload by applying sub-offload and super-offload transformations for creating subtasks and supertasks respectively. At L2, the incoming application is assumed to either not have been annotated or only partially annotated with offload directives. EOC carries out analyses to automatically insert #pragma soft-offload for code regions that can be identified as hot, parallel and conducive to offload, with the help of profile information. (This is a best-effort approach, not intended as a replacement for the developer). Apricot modules are leveraged for this purpose. Apricot's liveness and length analysis modules [42] are invoked to automatically insert and populate data transfer (i.e., in/out) clauses. This is followed by lowering of #pragma soft-offload, as in L1. Fig. 3 shows the overview of EOC architecture.

Directive #pragma soft-offload brings *relaxed semantics* to directive-based languages. It is intended to facilitate effective work division between the developer, compiler and runtime. Auxiliary clauses used with #pragma soft-offload

are identical to the ones used with #pragma offload. The use of auxiliary clauses is optional at L2. Fig. 2 shows the grammar for the new directive. Although the developer can mark arbitrary code regions with #pragma soft-offload, for the purpose of subtask and supertask creation, the marked code region is assumed to contain at least one hot parallel loop. This is the common case in typical applications.

EOC treats #pragma soft-offload as a hint and goes on to find code regions in the *neighbourhood* that can be good candidates for generating offloadable tasks. The first candidate is the code region $C$ marked with #pragma soft-offload itself (i.e., the task). In this case, #pragma soft-offload would simply be replaced by #pragma offload.

The second candidate is a code region containing a subloop obtained by splitting the index range of the main parallel loop in $C$ in two or more parts. We call this the *subtask*. Key challenges in subtask creation include: (i) identifying the split boundary for the parallel loop such that the execution time of the subloop on the accelerator is close to the execution time of the subloop on CPU, (ii) handling critical regions and reduction variables across the two subloops without additional synchronization, (iii) assessing profitability of subtask creation by ensuring that data transfer overheads do not offset gains from increased concurrency.

The third candidate is the code region containing $C$. This would either be a parent loop or a code region obtained by combining with a sibling loop. We call this the *supertask*. Formally, the entry block of the supertask dominates the entry block of the task and there is at least one path from the entry to the exit of the supertask that contains the exit block of the task. Multiple supertasks are possible for a given code region, EOC generates one per code region. Key challenges in supertask creation include: (i) creating supertasks that span across multiple functions, (ii) selecting the optimal supertask in presence of multiple options, (iii) assessing profitability by ensuring that the increase in serial code execution time on the accelerator does not offset gains from reduction in data transfer overheads.

# 3. COMPILER AND RUNTIME SUPPORT

In this section, we outline compiler and runtime support for addressing the aforementioned challenges.

## 3.1 Sub-offload

When a code region is offloaded to MIC, execution on the CPU is suspended until the offloaded code region executes to completion. This is an *either-or* model, where only one of the two processing units is active at a time, resulting in under utilization of resources. Also, the developer/runtime is forced to decide if a given code section should be offloaded to MIC or executed on the CPU. Ideally, a parallel code region should be able to execute on MIC *and* CPU concurrently.

LEO provides `signal` and `wait` clauses to enable asynchronous offload. Sub-offload splits the index range of the main parallel loop and generates two primary sub-loops: `R1` and `R2`. Sub-loop `R1` is asynchronously offloaded to MIC (as the subtask) while subloop `R2` executes on the CPU in parallel, as shown in Fig. 4(c).

For sub-offload transformation to be correct and profitable, a number of issues have to be taken care of. First, the split boundary should be selected in such a way that the time of execution of `R1` on MIC is close to the execution time

```
int g(..){
  ..
  for(i=0; i < K; i++){
    f();
    ..
  }
  #pragma omp parallel for
  for(j=0; j < M; j++){
    ..
  }
  ..
}

int f(..){
  ..
  #pragma soft-offload
      in(..) out(..)
  {
   #pragma omp parallel for
   /*hot loop*/
   for(i=0; i < N; i++){
     A[i] = B[i] * C[i];
     ..
   }
  }
  ..
}
```

**(a)**

```
int g(..){
  ..
  /* supertask */
  #pragma offload target(MIC)
        in(..) out(..)
  {
    for(i=0; i < K; i++){
      f();
      ..
    }
    #pragma omp parallel for
    for(j=0; j < M; j++){
      ..
    }
    ..
  }

int f(..){
  ..
  #pragma omp parallel for
  /*hot loop*/
  for(i=0; i < N; i++){
    A[i] = B[i] * C[i];
    ..
  }
  ..
}
```

**(b)**

```
int f(..){
  ..
  gettime(&t1);
  /*sample loop (c1 < 10) */
  for(i=0; i < c1; i++){
    A[i] = B[i] * C[i];
    ..
  }
  gettime(&t2);
  t_cpu = diff(t2,t1);
  r = K * (mem_ops/cpu_ops);
  c2 = (N + c1 * r)/(1 + r);

  if(cost_model(t_cpu, dsize, r_d, c2, c2){
    /* subtask (MIC loop R1) */
    #pragma offload target(MIC) in(..) out(..)
          signal(&f)
    {
      #pragma omp parallel for
      for(i=c1; i < c2; i++){
        A[i] = B[i] * C[i];
        ..
      }
    }
    /* main task (CPU loop R2) */
    #pragma omp parallel for
    for(i=c2; i < N; i++){
      A[i] = B[i] * C[i];
      ..
    }
    #pragma offload_wait target(MIC) wait(&f)
    {}
  }else{
    #pragma omp parallel for
    for(i=c1; i < N; i++){
      A[i] = B[i] * C[i];
      ..
    }
  }
}
```

**(c)**

**Figure 4: (a) Original code using soft-offload (b) Super-offload version (c) Sub-offload version**

of `R2` on CPU. This even distribution of work ensures minimum idle time for both CPU and MIC. Second, the transformation should be applied only if the speedup obtained by executing `R1` on MIC offsets the overheads of copying data into and out of MIC. Third, reduction variables and shared memory in critical sections should be correctly handled.

Sub-offload transformation splits the parallel loop in three parts. Let the upper bound of the loop be $N$ and the lower bound be 1 (without loss of generality). The index range $(1, N)$ is split into $(1, c_1), (c_1 + 1, c_2)$ and $(c_2 + 1, N)$. Three loops are created for each index range. The body of the original loop is copied into the three loops. The first loop is created for the purpose of *loop sampling*, which would

be used in assessing the profitability of this transformation based on runtime information (described in Section 3.1.2). The value of $c_1$ is a small constant. In the current implementation, $c1$ is set to 10. The second loop $(c_1 + 1, c_2)$ is offloaded to MIC asynchronously. The third loop $(c_2 + 1, N)$ executes on the CPU. We call the three loops: *sample loop*, *subtask* and *main task*, respectively.

Memory consistency across the two subloops is not a concern, since our index range splitting mechanism ensures that writes happen to disjoint memory regions. This is made possible by limiting sub-offload to loops with linear and contiguous memory accesses. Memory access pattern of a pointer is identified by parsing it as a function of the loop indices using *scalar evolution* representation [41]. For each loop index, the corresponding stride, lower bound and upper bound are extracted, from which the bounds of the pointer/array are obtained. This is done for all the pointers referenced in the subtask.

### 3.1.1 Identification of split boundary

In order to identify the split boundary (i.e. value of $c_2$), we need to estimate the ratio of the execution time of an iteration on MIC ($t_{mic}$) to the execution time of an iteration on CPU ($t_{cpu}$). The idea is that if one loop iteration takes twice as long to execute on CPU as compared to MIC, the index range of the subtask loop should be roughly twice the index range of the main task loop (for both loops to finish execution around the same time).

$$r = (N - c_2)/(c_2 - c_1) = t_{mic}/t_{cpu}$$
$$\implies c_2 = (N + r * c_1)/(1 + r)$$

We leverage Apricot's *cost-benefit analysis* module to estimate $r$. This module statically analyzes the loop and estimates of the number of CPU operations (*cpu_ops*) and memory operations (*mem_ops*) in the loop body. Since the loop may contain inner loops with non-constant upper-bounds, *cpu_ops* and *mem_ops* are generated as functions of inner loop iterations, and attain value at runtime.

The execution time of a loop iteration on MIC is a function of (i) the execution time of an iteration on CPU, (ii) the ratio of *mem_ops* to *cpu_ops*. From our experiments, we find that the ratio of execution time of one iteration on MIC to that on CPU is proportional to the ratio of *mem_ops* to *cpu_ops*. Intuitively, this is because MIC does better for compute intensive as compared to memory intensive code.

$$r = t_{mic}/t_{cpu} = f(K * (mem_ops/cpu_ops))$$

$K$ is the normalization factor obtained experimentally. The function $f$ bounds the value of $r$ between 0.25 and 4:

$$f(x) = \begin{cases} 0.25 & x < 0.25 \\ 4 & x > 4 \\ x & 0.25 < x < 4 \end{cases}$$

The equations for obtaining the value of $r$ and $c_2$ are inserted in the source code as part of the transformation. This is because $c_2$ is obtained from *mem_ops* and *cpu_ops*, which are determined at runtime (as mentioned before).

LEO clauses for asynchronous offload are inserted as part of sub-offload. Apricot modules for liveness and array-bound analysis are invoked for populating `in/out` clauses.

### 3.1.2 Loop sampling and runtime cost model

In order to determine profitability of sub-offload, we need to make sure that the time taken to copy data into and out of MIC for the subtask is offset by the gains obtained from concurrent execution.

Assuming that the subtask executes in parallel with the main task, the total execution time is equal to: *(execution time of sample loop + data transfer time for subtask + execution time of main task)*. Let *dsize* be the total size of the data variables that need to be copied into and out of MIC. Let $r_d$ be the data transfer rate. The total execution time is given by:

$$(t_{cpu} * c_1 + dsize/r_d + t_{cpu} * (N - c_2))$$

The execution time of the original unmodified loop is given by $(t_{cpu} * N)$. In order for this transformation to be profitable, the execution time of the transformed code should be less than that of the original code:

$$(t_{cpu} * N) > (t_{cpu} * c_1 + dsize/r_d + t_{cpu} * (N - c_2))$$
$$\implies t_{cpu} > (dsize/(r_d * (c_2 - c_1)))$$

This is the essence of the cost model function for this transformation. An *if* condition is inserted before the subtask to invoke the cost model function. The *then*-branch contains the subtask and main task. The *else*-branch contains the original loop (with index range $(c_1, N)$).

The value of *dsize* is obtained by adding up the size of the data variables specified in the `in/out` clauses for subtask. The value of $r_d$ (a constant) is obtained experimentally.

The value of $t_{cpu}$ is obtained using *loop sampling*. Timestamps ($t_1$ and $t_2$) are inserted before and after the sample loop, respectively. The value of $t_{cpu}$ is obtained at runtime from the two timestamps: $t_{cpu} = (t_2 - t_1)/c_1$. The sole reason for creating the sample loop is for obtaining the value of $t_{cpu}$, which would otherwise have to be estimated using static analysis. The idea is to peel out a few iterations of a loop (in the form of a *sample loop*) and use it to accurately estimate the execution time of one loop iteration at runtime.

**Discussion:** The value of $t_{cpu}$ obtained using loop sampling is an approximation to the actual value of $t_{cpu}$. Data locality and cache behavior can change during loop execution. Also, the presence of if-conditions and triangular loops can cause a variation in the execution time across iterations. A sampling-based approach would in general be more accurate than a static analysis or profiling based approach.

### 3.1.3 Critical regions and reduction variables

If a parallel loop does not contain reduction variables or critical sections, no variable renaming is necessary while splitting the loop. Since there is no order of execution among iterations in the original parallel loop, the two sub-loops (i.e., subtask and main task) can execute in parallel and perform read/write operations on private as well as shared memory. The manner in which data variables are handled by the offload operation in this case is similar to how OpenMP runtime manages thread-private and shared memory. By limiting sub-offload to loops with linear memory accesses, disjoint memory writes are ensured. However, in the presence of reduction variables and/or critical sections, additional work needs to be done to ensure correctness.

**Reduction Variables:** Given a reduction variable $v$ and reduction operation $\bigoplus$ in the original loop, we generate a new variable $v_mic$ and replace all occurrences of $v$ by $v_mic$ in the subtask. Code for computing $v = v \bigoplus v_mic$ is generated after the main task. The reduction variables and corresponding operators are identified from the `omp` clause. Fig. 5 shows the transformation for reduction variable `v`.

**Critical Sections:** A critical section is defined for the purpose of updating shared memory that must not be concurrently accessed by more than one thread. Since we execute two loops in parallel, each of which has a separate OpenMP runtime, the semantics of a critical section are violated. Consider a parallel loop with a shared array $q$ that is updated inside a critical section. When the loop is split, the two loops would have separate copies of array $q$ and would both update their own copies. However, when the subtask finishes, it's copy of array $q$ would be copied back to the host memory in the same location as CPU loop's $q$. As a result, the updates from one of the two loops would be lost (depending on which of the two loops finishes first).

In order to preserve the semantics of a critical section, we need to (i) rename shared memory variables being accessed in the critical section, (ii) generate code for merging the updates from the two loops. Fig. 5 shows the transformation.

For each critical section, we identify the shared memory variables being updated in the critical section. This is done with the help of gcc's *memory tag* infrastructure, which keeps track of read and write operations to all memory locations. Given a shared memory variable *smv*, we generate a new variable *smv_mic* and replace all occurrences of *smv* by *smv_mic* in the MIC loop. In order to generate code for merging *smv_mic* and *smv*, the correct merge logic needs to be identified. In the case of reduction variables, merging is trivial. This is because the reduction operator is known and the merge logic is to simply apply the reduction operator to the two scalar variables. This is not the case for shared memory variables in critical sections.

**Horizontal and vertical integration:** Following observations motivate us to attempt merging of shared memory for critical sections: (i) every write operation in a critical section is typically performed by multiple threads/iterations (ii) critical sections are usually small and have simple control flow. We screen out critical sections that don't conform to the observations above (e.g., contain conditional writes).

A write to a shared memory location can be classified in two categories: (A) one that updates the memory location, (B) one that writes a fresh value to a memory location. The case of memory update can be further sub-divided into two categories: (i) where the memory update happens through a single operator that is commutative and associative, (ii) where multiple operators are involved or the operator(s) is not commutative/associative. The case of fresh value write can be sub-divided into two categories: (i) one where the value being written is not read from the shared memory, (ii) one where the value being written contains content read from shared memory. In current EOC design, we are able to handle cases A(i) and B(i), through *horizontol integration* and *vertical integration* respectively, as described next.

Visualize two non-overlapping memory regions *smv* and *smv_mic* of same or different lengths. In horizontal integration, the two memory regions are merged by placing one on top of the other. The values $smv[i]$ and $smv_mic[i]$ are combined using a merge operator $\bigodot$ that is commutative and associative (e.g., multiplication). This follows from A(i). Following code is generated for horizontal integration:

```
..
#pragma omp parallel for reduction(+:v)
for(i=0; i < N; i++){
  ..
  v += A[i];
  ..
  #pragma omp critical
  {
    for (j = 0; j < NQ; j++)
      q[j] += qq[j];
  }
  ..
  #pragma omp critical
  {
    smv->x = x;
    smv->y = y;
    smv++;
    smv_len++;
  }
  ..
}
..
```

**(a)**

```
..
if(cost_model(..)){
  ..
  v_mic = 0; //1 for multiplication
  q_mic = clone(q);
  smv_mic = clone(smv);
  smv_len_mic = 0;
  /* subtask (MIC loop) */
  #pragma offload..inout(q_mic:length(NQ)) inout(smv_mic:
      length(max)) inout(smv_len_mic,v_mic) signal(&f)
  {
    #pragma omp parallel for reduction(+:v_mic)
    for(i=c1; i < c2; i++){
      ..
      v_mic += A[i];
      ..
      #pragma omp critical
      {
        for (j = 0; j < NQ; j++)
          q_mic[j] += qq[j];
      }
      ..
      #pragma omp critical
      {
        smv_mic->x = x;
        smv_mic->y = y;
        smv_mic++;
        smv_len_mic++;
      }
      ..
    }
  }
  /* main task (CPU loop) */
  #pragma omp parallel for reduction(+:v)
  for(i=c2; i < N; i++){
    ..
    v += A[i];
    ..
    #pragma omp critical
    {
      for (j = 0; j < NQ; j++)
        q[j] += qq[j];
    }
    ..
    #pragma omp critical
    {
      smv->x = x;
      smv->y = y;
      smv++;
      smv_len++;
    }
    ..
  }

  #pragma offload_wait target(mic) wait(&f)
  {}

  /* adding reductions */
  v += v_mic;
  /* horizontol integration */
  for(i=0; i < NQ; i++){
    q[i] += q_mic[i];
  }
  /* vertical integration */
  for(i=0; i < smv_len_mic; i++){
    smv[smv_len + i] = smv_mic[i];
  }
  smv_len += smv_len_mic;
  ..
}else{
  ..
}
..
```

**(b)**

**Figure 5: Handling reduction variables and critical regions: (a) Original code (b) Transformed code**

```
for(i=0; i < length(smv); i++){
  smv[i] = smv[i] ⊙ smv_mic[i];
}
```

In vertical integration, the two memory regions are merged by appending one to the other. The reasoning is as follows. If a write operation is performed by multiple threads/iterations, the value being written is fresh and not read from shared memory, then there are two possible scenarios: (i) multiple threads/iterations write to the same memory location, in which case only the last write is important, (ii) writes are performed on different memory locations. Given that there is no order of execution among threads and the writes are uniform/unconditional, no additional support is needed for scenario (i): the shared memory region is not cloned and/or renamed on the coprocessor side. For non-uniform/conditional writes, EOC will bail out. For scenario (ii): if the writes are independent, critical section is redundant. This case makes little sense in practice and can be easily handled by taking the memory region out of the critical section. Finally, when uniform writes are performed on different memory locations inside a critical section, such that they are not independent, we have the case of dynamic data structures, such as linked lists. Vertical integration is applied to those cases (*smv_mic* is appended to *smv* in the end) as shown in the following code:

```
for(i=0; i < current_length(smv_mic); i++){
  smv[current_length(smv) + i] = smv_mic[i];
}
current_length(smv) += current_length(smc_mic);
```

All reads and writes are tracked using gcc's memory tag infrastructure. Candidates for horizontol integration are identified by looking for read and write to the same memory location using a commutative and associative operator. Candidates for vertical integration are identified by looking for fresh value writes to a memory location, such that the rhs has no data dependence on any shared memory variables. The merge operator $\odot$ for horizontol integration is obtained by parsing the memory update statement in the critical section IR for a given shared memory variable. The length variable, needed for vertical integration, is obtained by parsing all scalar update statements in the critical section of the form $len = len + k$ and picking the one that has a one-to-one relationship with the fresh-value write for the given memory region. This is a heuristic-based approach.

**Discussion:** We have not investigated updates to complex pointers, multi-dimensional arrays or indirect array accesses in critical sections. In the current implementation, the compiler only handles critical sections that update single-level pointers and one-dimensional arrays and have no conditional writes. If the compiler is not able to detect the merge operator for horizontal integration or the current length of pointers for vertical integration, it bails out. The current implementation also does not handle cases where both horizontal and vertical integration may be needed for a shared memory variable. We err on the conservative side to preserve correctness. The dual of correctness is coverage, which will be improved incrementally in subsequent versions of EOC. We are currently implementing a heap linearization mechanism for handling multi-dimensional arrays and multi-level pointers.

**Algorithm 1** Sub-offload Algorithm

**Input:** Code region (parallel loop $L$ with body $B$), index range $(1, N)$; constants $K$, $r_d$;
**Procedure** Sub-offload()
  /* *index range splitting* */
  split index range $(1, N)$ in three $I_1$=$(1, c1)$, $I_2$=$(c1 + 1, c2)$ and $I_3$=$(c2, N)$
  generate loops $L_1$, $L_2$ and $L_3$ with index ranges $I_1$, $I_2$ and $I_3$
  copy body $B$ into $L_1$, $L_2$ and $L_3$
  invoke liveness, array-bound analysis on $L_2$; generate LEO clauses for $L_2$
  /* *loop sampling* */
  insert timestamps $t1$, $t2$ before and after $L_1$
  insert `t_cpu = time_dif(t1, t2)/c1` before $L_2$
  invoke cost-benefit analysis phase on $L$; obtain `cpu_ops` and `mem_ops`
  /* *split boundary identification* */
  obtain `dsize` from `in/out/inout` clauses
  insert `cpu_ops`, `mem_ops` and `dsize` before $L_2$
  insert `r = K * (mem_ops/cpu_ops)` before $L_2$
  insert `{if(r < 0.25) r = 0.25 else if(r > 4) r = 4}` before $L_2$
  insert `c2 = (N + c1 * r)/(1 + r)` before $L_2$
  /* *main transformation* */
  create `if` condition with predicate `cost_model(t_cpu, dsize,` $r_d$`, c1, c2)`
  move $L_2$ and $L_3$ in `then` branch
  insert LEO clauses for `wait` after $L_3$
  generate loop $L_4$ with body $B$ and index range $(c_1, N)$ in `else` branch
  /* *handling reduction vars* */
  identify reduction variables V, operators O from `omp for`
  **for** $(v_i \in \mathcal{V})$ and $(\bigoplus \in \mathcal{O})$ **do**
    rename reduction variable $v_i$ to $v_mic_i$ in $L_2$ and `in/out` clauses
    insert $v_i = v_i \bigoplus v_mic_i$ after `wait` clause
  **end for**
  /* *handling critical sections* */
  identify `omp critical` sections $\mathcal{C}$
  **for** $(cr_i \in \mathcal{C})$ **do**
    identify shared memory variables $\mathcal{SMV}$ being updated
    **for** $(smv_j \in \mathcal{SMV})$ **do**
      copy $smv_j$ into $smv_mic_j$
      rename $smv_j$ to $smv_mic_j$ in $L_2$ and `in/out` clauses
      $H$ = (check for read,write to $smv_j$ in same stmt in $cr_i$; identify update operator $\odot$)
      **if** $(H)$ **then**
        insert {`for(i=0; i < length(`$smv_j$`); i++)` $smv_j[i] = smv_j[i] \odot smv_mic_j[i]$} after `wait` clause
      **else**
        $V$ = (check for pointer operations and length variable $l$ in $cr_i$)
        **if** $(V)$ **then**
          rename $l$ to $l_mic$ in $L_2$ and `in/out` clauses
          insert {`for(i=0; i <` $l_mic$`; i++)` $smv[l+i] = smv_mic[i]$}; {$l = l + l_mic$} after `wait` clause
        **else**
          abort; exit(0)
        **end if**
      **end if**
    **end for**
  **end for**

## 3.2 Super-offload

Data transfer overheads can easily nullify the gains from increased parallelism. The goal of super-offload is to reduce data transfer overheads while creating offloadable tasks, beyond what can be achieved with data reuse optimizations. The basic idea is to consolidate work.

Fig. 4(a) shows a hot parallel loop contained inside function `f`. The developer would mark this loop with a #pragma soft-offload. If the hot parallel loop were to be identified as the code region for offload, it would result in the creation of multiple offloadable tasks (one per iteration of the loop in `g`). However, if the serial loop in `g` were identified as the offload candidate, only one offload task would be created for the entire loop, resulting in fewer data transfers (once per loop as compared to once per iteration).

This, however, may lead to increased serial code execution on the accelerator and cause a slowdown; depending on what fraction of the time spent in the outer loop is spent executing serial code. If the parallel code regions constitute a small fraction of the total execution time of the outer loop, creating a supertask may not be beneficial.

It is often beneficial to consolidate and include parallel sibling loops within a supertask. As shown in Fig. 4(b), the supertask includes the parallel sibling loop next to the serial loop in `g()`. The primary advantage of consolidating neighbouring loops and executing them together on the accelerator is that it eliminates data transfers (between CPU

**Algorithm 2** Super-offload Algorithm

**Input:** Code region $L$; profile information; constants $c$, $r_d$
**Procedure** Super-offload()
  $k = 0$;
  **while** (1) **do**
    $P$ = parent_loop($L$)
    **if** ($P$) **then**
      $t_p$ = serial_execution_time_from_profile($P$)
      $dsize_L$ = data_transfer_size_estimate($L$)
      $dsize_P$ = data_transfer_size_estimate($P$)
      **if** (supertask_profitable($dsize_L$, $dsize_P$, $t_p$, $c$, $r_d$)) **then**
        $supertask[k] = P$
        $k$++
        continue
      **end if**
    **end if**
    $L1$ = parallel_sibling_loop($L1$)
    **if** ($L1$) **then**
      **if** (data_dependence($L$, $L1$)) **then**
        $supertask[k]$ = create_outer_block($L$, $L1$)
        $k$++
        continue
      **end if**
    **end if**
  **end while**
  $final_supertask = supertask[k]$
  invoke liveness, array-bound analysis on $final_supertask$
  generate LEO clauses for $final_supertask$
**Input:** Data sizes $dsize1$, $dsize2$; Time $t_s$; constants $c$, $r$;
**Output:** true or false
**Procedure** supertask_profitable()
  $dsize_reduction = dsize1 - dsize2$
  **if** ($dsize_reduction/r > c * t_s$) **then**
    return true
  **else**
    return false
  **end if**

and accelerator) resulting from data dependencies.

Super-offload is performed iteratively for each code region. At each step of the iteration, EOC identifies the parent loop and parallel sibling loop(s) for the code region marked by #pragma soft-offload. An outer block is created for consolidating the sibling loops (when applicable) followed by hoisting of the #pragma soft-offload to the parent loop when profitable (as determined by our cost-model). This process repeats iteratively. At the end of each iteration, the supertask from that iteration is added to the set. Supertask from the last iteration is usually the most profitable one. We find that most applications allow opportunities for one/two-step supertask creation beyond which it is usually either not applicable or not profitable.

### 3.2.1 Auto-inlining

Since EOC carries out analysis over gcc IR and applies transformations to the source code, interprocedural optimization becomes harder. In order to perform analyses across functions and facilitate super-offload, we have developed a component for auto-inlining. The goal is to find function invocations inside offloadable code regions and mark the corresponding functions with `__always_inline` flag. The algorithm proceeds as follows: the body of each function is inspected to find function invocations. If function invocation `foo()` is within a hot loop `l`, function `foo` is marked as a potential candidate for inlining. If `l` is marked by #pragma omp parallel, or function `foo` contains a hot loop marked by #pragma omp parallel, `foo` is marked for inlining.

### 3.2.2 Auto-profiling

We have interfaced EOC with *gprof* [4] in order to obtain profile information. The profiler is run early in the analysis. Profile information is processed and organized per function, per loop and can be queried on-demand. Profile information is used to screen out parallel loops that take less than `t%` ($t < 10$) of the total execution time and are, therefore, not legitimate candidates for offload. Super-offload uses profile information to assess profitability, as described next.

```
int g(..){
  ..
  for(i=0; i < K; i++){
    f();
    ..
  }
  #pragma omp parallel for
  for(j=0; j < M; j++){
    ..
  }
  ..
}

int f(..){
  ..
  /* hot loop */
  #pragma omp parallel for
  #pragma acc soft-kernels
   for(i=0; i < N; i++){
     A[i] = B[i] * C[i];
     ..
  }
  ..
}
```

**(a)**

```
int g(..){
 ..
 #pragma acc kernels copy(..)
 for(i=0; i < K; i++){
   f();
   ..
 }
 #pragma omp parallel for
 for(j=0; j < M; j++){
   ..
 }
 ..
}

int f(..){
 ..
 /* hot loop */
 #pragma acc parallel loop
       gang(..)
  for(i=0; i < N; i++){
    A[i] = B[i] * C[i];
    ..
 }
 ..
}
```

**(b)**

```
int f(..){
  ..
  gettime(&t1);
  /*sample loop (c1 < 10) */
  for(i=0; i < c1; i++){
    A[i] = B[i] * C[i];
    ..
  }
  gettime(&t2);
  t_cpu = diff(t2,t1);
  r = K * (mem_ops/cpu_ops);
  c2 = (N + c1 * r)/(1 + r);

  if(cost_model_acc(t_cpu, dsize, r_d, c2, c2)
      {
    /* subtask (GPU/MIC loop R1) */
     #pragma acc parallel loop copy(..)
      gang(..) async(k)
     for(i=c1; i < c2; i++){
       A[i] = B[i] * C[i];
       ..
     }
    /* main task (CPU loop R2) */
    #pragma omp parallel for
    for(i=c2; i < N; i++){
      A[i] = B[i] * C[i];
      ..
    }
    #pragma acc wait(k)
    ..
  }else{
    #pragma omp parallel for
    for(i=c1; i < N; i++){
      A[i] = B[i] * C[i];
      ..
    }
  }
}
```

**(c)**

**Figure 6: OpenAcc: (a) Original code using soft-kernels (b) Super-offload version (c) Sub-offload version**

### 3.2.3 Compile-time cost model

Many-core accelerators have low serial performance as compared to the multi-core CPU. The serial performance of MIC, for example, is always less than half and often less than one-third the serial performance of a Xeon multi-core processor. Higher number of cores and better vectorization capabilities provide increased parallelism and make up for the loss in serial performance for parallel workloads.

Super-offload typically leads to an increase in serial code execution on the accelerator and a decrease in the amount of data transferred over PCIe. The former causes a slowdown, while the later improves performance. Hence a cost model is needed to assess profitability of supertask creation.

For a given supertask, let *ts* be the time spent executing serial code on the CPU. The serial execution time on MIC

would be *c*ts*, where $c > 1$. Let $d$ be the total amount of data transfer reduction due to super-offload and $r$ be the rate of data transfer over PCIe. Super-offload would be profitable if $d/r > c*ts$ (i.e., gains from data transfer reduction outweigh the losses from serial code execution). We conservatively assume that the serial performance of MIC is one-fifth that of the CPU, and set $c = 5$. The value of *ts* is obtained from profile information; $r$ is a constant measured offline and $d$ is estimated by inspecting the data variables in *in/out* clauses. This cost model allows us to assess the profitability of supertask creation and select the most profitable one when there are multiple candidates.

## 3.3 Auto-tuning

We have implemented a basic auto-tuner as part of EOC. For code regions for which both subtask and supertask are generated, the auto-tuner selects the more profitable one. This is done by executing both subtask and supertask for varying input data sizes and selecting the one that runs faster. In most cases, the selection is independent of the input data size. For cases where the selection depends on the input data size, the auto-tuner simply outputs a message for the developer indicating so. We plan to extend this component to generate multi-versioned code that selects the optimal version based on input data size at runtime.

## 3.4 Other directive-based languages

The directive introduced in this paper can be applied to other directive-based dialects as well, with suitable modifications to the front end and cost model library. Figure 6 shows an example for OpenAcc. Directive `#pragma acc soft-kernels` is equivalent to the soft-offload directive. Super-offload and sub-offload versions for OpenAcc are similar to the ones generated for LEO (Fig. 4). Depending on the amount of serial code in function f(), #pragma acc data can also be used instead of #pragma acc kernels in the outer loop. Since OpenAcc is not device-centric, our solution can potentially be used across multiple accelerator types.

# 4. EVALUATION

EOC framework has been implemented on top of `gcc` and uses Apricot modules for insertion of #pragma offload and in-/out clauses. Sub-offload and super-offload are implemented as compiler optimizations and can be invoked independently. They are invoked together as part of #pragma soft-offload lowering at *L1*. The resulting output is optimized source code that can be further modified and compiled with `icc`.

We evaluate EOC with respect to Intel Xeon Phi (MIC). The experiments are carried out on a Xeon E5-2660 server equipped with a MIC card and the necessary software. Xeon E5-2660 has 8 cores, each running at 2.2GHz with 2 threads per core. MIC has 60 cores each running at 1.05GHz with four threads per core, a total of 32MB L2 cache and 8GB GDDR5 memory. We present results for NAS Parallel (NPB-2.3 in C), SpecOMP and Phoenix benchmark suites. EOC is invoked on these benchmarks and the optimized source code is compiled with `icc composer_xe_2013.1.117` at `-O3`.

## 4.1 Results

We experimented with a total of 14 benchmarks: all the eight benchmarks in NAS (`FT`, `MG`, `EP`, `SP`, `BT`, `CG`, `IS`, `LU`); the three C benchmarks in SpecOMP (`320.equake`, `330.art`, `332.ammp`); a Fortran benchmark (`312.swim`) and two from Phoenix (K-means and MatrixMultiply). These benchmarks have parallel loops and code regions annotated with OpenMP. For some of the benchmarks, code modifications are made to convert multi-dimensional arrays to one-dimensional arrays.

Performance gains were observed for 11 of these benchmarks. The results are summarized in Fig. 7. There are two baselines for our experiments: (i) the unmodified version of each benchmark executed on the CPU (ii) unoptimized CPU-MIC version obtained by inserting #pragma offload for the hot and parallel loops in each benchmark.

In order to obtain the optimized CPU-MIC version for a given benchmark, all occurences of #pragma offload are replaced by #pragma soft-offload and EOC is invoked. When both super-offload and sub-offload are applicable for a benchmark, multiple output files are generated by EOC. Fig. 7(a) shows the performance gains obtained with sub-offload and super-offload independently. Baseline is the unoptimized CPU-MIC version. Super-offload triggers for five benchmarks with the speedup ranging between 2.8x and 24x. Sub-offload triggers for seven benchmarks with speedups ranging from 1.2x to 11x. For three of the benchmarks (320.equake, MG, BT) super-offload and sub-offload trigger together.

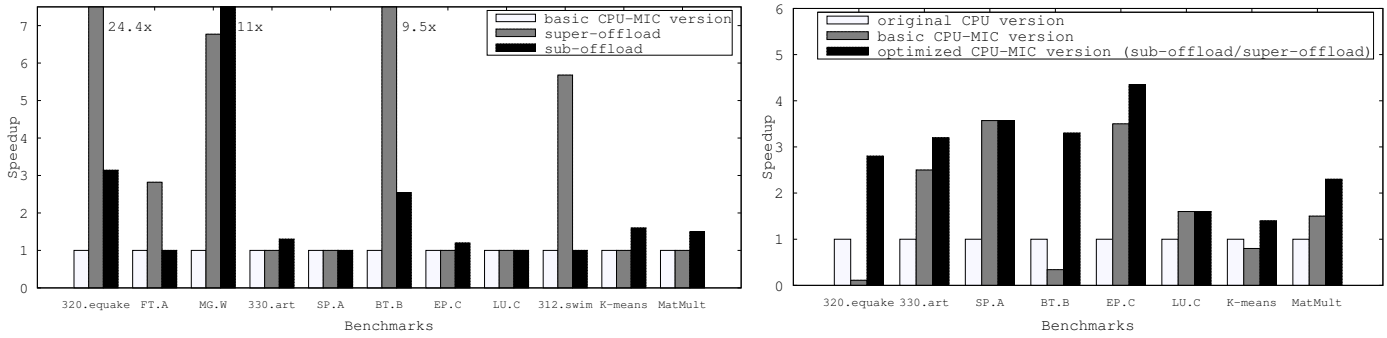
Fig 7(b) compares the performance of the original CPU version (as baseline) with optimized CPU-MIC version obtained by applying sub-offload and super-offload. In cases where both supertask and subtask creation trigger for the same code region, the best performing version is picked by the auto-tuner. Performance improvement is observed for eight benchmarks, with the gains ranging between 1.4x-4.4x.

The performance of unoptimized CPU-MIC version is also shown. For several benchmarks, unoptimized CPU-MIC version performs worse than the CPU version but significantly better when optimized. This underscores the need of the approach described in this paper. Three benchmarks (MG, FT, 312.swim) continue to perform worse as compared to the CPU-only version even when optimized. This is due to high data transfer overheads.

**Impact of input datasize:** We tried different input datasets/classes in NAS Parallel benchmarks. For small input datasets (class=W, S), no gains are observed with MIC, with or without our approach. This is due to the the small execution time (and hence limited work) for small input datasets.

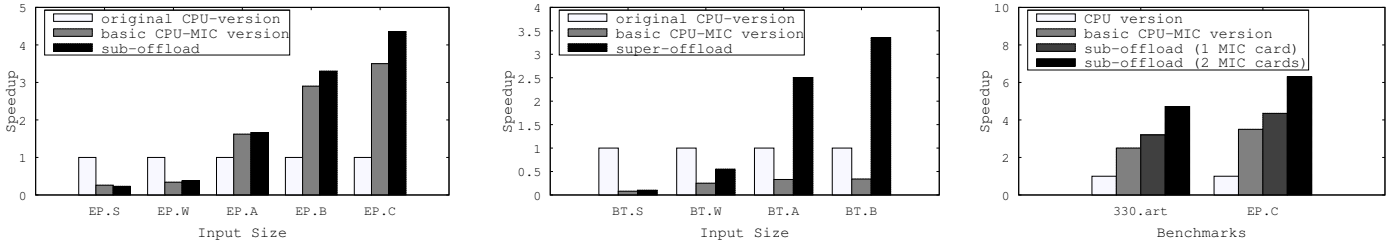
For very large datasets (class=C), out-of-memory errors are often encountered during offload, with or without our approach. This is due to the limited memory on MIC. In general, we find that as input data size increases, the gains improve, until we run out of memory. Fig. 8 shows the performance of benchmarks EP and BT for the different input datasets ($W < S < A < B < C$).

**Multiple MIC cards:** The current implementation of EOC only targets one MIC accelerator per CPU. It is possible to extend sub-offload to target multiple MIC cards by creating one subtask for every MIC card. In order to do a feasibility test, we added a second MIC card to our server and hand-modified benchmarks 330.art and EP to create multiple subtasks per loop (one for each MIC card). Fig. 8(c) shows the results. As compared to the CPU-only version, we observe a speedup of 3.1x with one MIC card and 4.7x with two MIC cards, for 330.art. For EP, we get 4.35x and 6.3x respectively. The second generation of EOC would support multiple MIC cards.




**(a)** **(b)**

**Figure 7: (a) Performance gains obtained with super-offload and sub-offload. Baseline is unoptimized CPU-MIC version (b) Performance of optimized CPU-MIC version (obtained by applying sub-offload/super-offload) vs. original CPU version vs. unoptimized CPU-MIC version**




**(a)** **(b)** **(c)**

**Figure 8: Impact of varying input data size on performance of (a) EP (b) BT. Higher gains with increasing data size. (c) Performance of sub-offload with multiple MIC cards for EP and 330.art**

**Overheads:** We measured the overheads of merging reduction variables and critical sections as well as runtime cost model across all benchmarks. As expected, all dynamic overheads are negligible– less than 0.0001% for the runtime cost model; less than 0.004% for merging reduction variables and critical sections.

**Compile time:** The compile time varies between 12 seconds (for `EP`) and 220 seconds (for `BT`). It depends on the number of lines in the benchmark, the code structure, and the number of optimizations applied. The profile time is equal to the execution time for the given input dataset. The auto-tuning time is the sum of the execution time of all the generated source code versions.

**K-sensitivity:** The module that identifies the split boundary for sub-offload uses a constant `K` in the cost model. This constant is obtained experimentally and is set to 0.3 in the current implementation. Fig. 9 shows how the performance of the benchmarks varies with different values of `K` ranging between 0.1 and 0.5. The execution time is normalized by the execution time of the benchmarks for K=0.3.

## 4.2 Analysis

We briefly analyze the benchmarks shown in Fig. 7(b)). **320.equake**: There are two hot and parallel loops within an outer serial loop that are conducive for offload. One of the two loops is the main loop, for which both sub-offload and super-offload are applicable and profitable. Due to high data transfer overheads, super-offload provides better gains. The outer serial loop is identified as the supertask. Most of the time spent in this loop is spent executing the two parallel

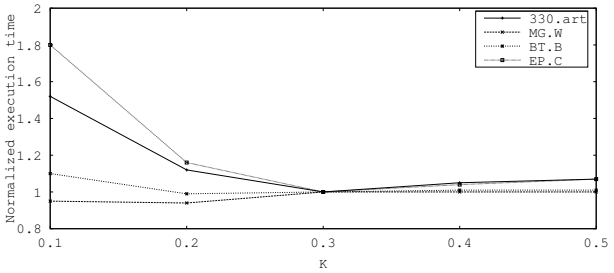



**Figure 9: K-sensitivity: normalized execution time of benchmarks for different values of K. Execution time is 1 for K=0.3. Lower is better.**

loops. The unoptimized CPU-MIC version (with the two hot and parallel loops as offload candidates) runs slower than the CPU version, due to high data transfer overheads. The supertask version, with the outer serial loop as the offload candidate, yields 2.8x speedup over the CPU version.

**330.art**: Has an omp parallel region containing a hot parallel loop that is conducive to offload. There are no outer or sibling loops for applying super-offload. The unoptimized CPU-MIC version (that offloads the parallel region to MIC) runs 2.5x faster than the CPU-only version. After applying sub-offload to this parallel code region, the speedup increases to 3.2x for one MIC card and 4.7x for two MIC cards. The parallel loop has a critical region, which is handled correctly.

**EP**: There is an omp parallel region with a hot parallel loop inside that is conducive to offload. There are no outer or sibling loops for applying super-offload. The unoptimized CPU-MIC version (that offloads the parallel region to MIC) runs 3.5x faster than the CPU-only version. After applying sub-offload to this parallel code region, the speedup increases

to 4.35x for one MIC card and 6.3x for two MIC cards. The parallel loop has reduction variables and a critical region inside, which are handled correctly by EOC.

**SP**: Has a large omp parallel region with a number of small parallel loops inside. Sub-offload is not applicable, as the execution time gets divided among the large number of parallel loops. The parallel region contains the main serial loop, making super-offload unapplicable/unprofitable. The unoptimized CPU-MIC version (that offloads the entire parallel region to MIC) runs 3.5x faster than the CPU version.

**BT**: Most of the time is spent inside a function that has a hot parallel loop. It gets repeatedly invoked from another function inside a serial loop. The unoptimized CPU-MIC version (that offloads the parallel loop to MIC) runs slower than the CPU version. Both super-offload and sub-offload are applicable to this loop. Due to high data transfer overheads, super-offload is more effective and runs 9.5x faster than the unoptimized CPU-MIC version and 3.3x faster than the CPU version.

**LU**: Has a large omp parallel region containing a number of parallel loops inside. Sub-offload is not profitable, as the execution time gets divided among the numerous parallel loops. The parallel region contains the main serial loop, making super-offload unapplicable/unprofitable. The unoptimized CPU-MIC version (that offloads the entire parallel region to MIC) runs 1.6x faster than the CPU version.

**K-means and MatMult**: These are map-reduce benchmarks taken from the Phoenix benchmark suite. The parallel for loop in the map function is a candidate for offload. For K-means, the unoptimized CPU-MIC version (that offloads the entire loop to MIC) runs slightly slower than the CPU version. Application of sub-offload results in a speedup of 1.4x for medium-sized datasets. For MatMult, the unoptimized CPU-MIC version is 1.5 times faster than the CPU version. The speedup improves to 2.3x with sub-offload. There are no oportunities for supertask creation.

# 5. DISCUSSION AND RELATED WORK

Numerous compilers [34, 18, 29, 47, 32, 36, 37, 11, 24, 46, 10] and productivity tools [48, 45, 39] have been developed to minimize programming effort and optimize performance for GPGPUs in the context of Cuda and OpenCL. Despite an impressive body of work, the problem of code homogeneity, readability, longevity and maintenance remain concerns for heterogeneous computing. This has been recognized and efforts have been launched to simplify and unify accelerator programming with the help of directive-based languages. OpenAcc [7] is an outcome of this recognition. Notable contemporary efforts include LEO, OpenHMPP [1], OmpSs [6], Offload C++ [25], OpenMPC [33], etc. These languages provide directives for specifying offloadable tasks and orchestrating data transfers efficiently. Some of these initiatives, such as StarPU [15], also support heterogeneous task scheduling at runtime.

LEO, OpenAcc and other similar initiatives are in a state of evolution. Our approach is to evaluate them with respect to existing and upcoming coprocessor architectures, identify limitations and propose solutions that can be easily integrated. Intel MIC is an important step in the evolution of coprocessor architectures, and relatively less explored.

Task graphs have been leveraged in numerous contexts including iteration scheduling for multicore processors [21, 19], thread assignment for heterogenous cores [20, 38], process scheduling for clusters [27, 15], task scheduling for heterogeneous processors [40, 15, 26, 44], etc. In the current form, directive-based languages allow scheduling and mapping of task graphs at runtime (as in StarPU and OmpSs), but do not facilitate application of task graph optimizations. Once an offloadable task has been specified by the developer, there is no provision for task restructuring. The directive introduced in this paper allows task graph optimizations to be selectively applied to code regions marked by the directive.

Super-offload extends and generalizes the offload hoisting optimization in Apricot [42]. Unlike offload hoisting, super-offload is applicable across functions; supports combining of sibling loops into larger code blocks and has a generalized algorithm and cost model. Kernel-splitting in OpenMPC [33] divides the body of the loop into multiple parts in order to handle synchronization constructs (horizontol splitting). Sub-offload splits the index range of the loop into multiple parts each with the same loop body (vertical splitting). The problem of distributing the kernel among multiple GPUs has been studied in the context of OpenCL kernels [31]. Kim et al [31] use kernel sampling to obtain memory access range of each GPU for the kernel, while EOC uses sampling to estimate the execution time of a loop iteration on the CPU. Index range splitting [30] is a classic compiler optimization that is applied for the purpose of improving data locality and enabling other optimizations, such as tiling [16]. Sub-loops execute *serially* one after another. We extend this transformation and combine it with asynchronous offload to enable concurrent execution of loops on CPU and coprocessor.

Partitioning of parallel loops [17, 14] has been explored in the context of distributed shared-memory multiprocessors. The main focus was optimal data partitioning and minimization of communication overheads. Our work is related to and benefits from some of these ideas. Since CPU and MIC do not share memory and there is no explicit communication between them while the two subloops execute concurrently, the issues that need to be addressed are quite different.

Partitioned global address space (PGAS) family of languages, such as X10 [43], Chapel [23], Phalanx [28], etc also support GPU programming.They provide a uniform abstraction for cluster and accelerator programming. PGAS languages and directive-based languages are likely to co-exist and target developers with different levels of expertise.

# 6. CONCLUSIONS

We present a new directive to add *relaxed semantics* to directive-based languages and facilitate offloadable task restructuring. The goal is to enhance collaboration between the developer, compiler and runtime. We present a framework called EOC that supports this directive for LEO and Intel Xeon Phi (MIC). Central to our approach is the idea of sub-offload and super-offload, loop sampling and use of runtime information. Sub-offload and super-offload can be applied independently as compiler optimizations. Only one directive is added to preserve economy of mechanism and minimize external complexity. Evaluation is carried out with respect to NAS Parallel, Phoenix and SpecOMP benchmarks. Speedups range between 1.3x-4.4x with the parallel CPU version as baseline and 1.2x-24x with the unoptimized CPU-MIC version as baseline. The solution, with suitable modification, can be applied to some of the other directive-based languages as well. We plan to add support for OpenAcc in the near future.

# 7. REFERENCES


[1] Caps openhmpp. http://www.caps-entreprise.com/openhmpp-directives.

[2] Cuda. http://www.nvidia.com/object/cuda_home_new.html.

[3] Gnu compiler collection. http://gcc.gnu.org.

[4] Gnu gprof. http://sourceware.org/binutils/docs/gprof.

[5] Intel c++ compiler. http://www.intel.com/Compilers.

[6] Ompss. http://pm.bsc.es/ompss.

[7] Openacc: Directives for accelerators. http://www.openacc-standard.org/.

[8] Opencl. https://developer.nvidia.com/opencl.

[9] The openmp api. http://www.openmp.org.

[10] The portland group (pgi). http://www.pgroup.com.

[11] R-stream compiler. https://www.reservoir.com/rstream.

[12] Threading building blocks. http://software.intel.com/en-us/articles/intel-tbb.

[13] Top 500 supercomputers. http://www.top500.org.

[14] A. Agarwal, D. A. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 6(9):943–962, 1995.

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2), 2011.

[16] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[17] R. Barua, D. A. Kranz, and A. Agarwal. Communication-minimal partitioning of parallel loops and data arrays for cache-coherent distributed-memory multiprocessors. In *LCPC*, pages 350–368, 1996.

[18] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Compiler Construction (CC)*, pages 244–263, 2010.

[19] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2009.

[20] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers (CF)*, 2006.

[21] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, 1995.

[22] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of gpu clusters with ompss. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[23] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *International Journal of High Perf. Comput. Appl.*, 21(3), 2007.

[24] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *PACT*, 2010.

[25] A. F. Donaldson, U. Dolinsky, A. Richards, and G. Russell. Automatic offloading of c++ for the cell be processor: A case study using offload. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 901–906, 2010.

[26] M. M. Eshaghian and Y. C. Wu. Mapping heterogeneous task graphs onto heterogeneous system graphs. In *Proceedings of the 6th Heterogeneous Computing Workshop (HCW)*, 1997.

[27] D. et al. Scheduling parallel task graphs on (almost) homogeneous multicluster platforms. *IEEE Trans. Parallel Distrib. Syst.*, 20(7), 2009.

[28] M. Garland, M. Kudlur, and Y. Zheng. Designing a unified programming model for heterogeneous machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[29] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic cpu-gpu communication management and optimization. In *PLDI*, pages 142–151, 2011.

[30] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[31] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP)*, pages 277–288, 2011.

[32] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS)*, 2012.

[33] S. Lee and R. Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[34] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel programming (PPoPP)*, 2009.

[35] S. Lee and J. S. Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[36] J. Lee et al. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, pages 193–204, 2010.

[37] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[38] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II (Euro-Par)*, 1996.

[39] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS*, 2013.

[40] J. A. Pienaar, S. Chakradhar, and A. Raghunathan. Automatic generation of software pipelines for heterogeneous parallel systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[41] S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *In 2005 International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 218–232, 2005.

[42] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar. Apricot: an optimizing compiler and productivity tool for x86-compatible many-core coprocessors. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS)*, pages 47–58, 2012.

[43] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[44] A. Sbirlea, Y. Zou, Z. Budimlic, J. Cong, and V. Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. In *Proceedings of the 13th ACM SIGPLAN/SIGBED LCTES*, 2012.

[45] J. Sim, A. Dasgputa, H. Kim, and R. Vuduc. A performance analysis framework for identifying performance benefits in GPGPU applications. In *Proc. ACM Symp. Principles and Practice of Parallel Prog. (PPoPP)*, 2012.

[46] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[47] Y. Yang et al. A gpgpu compiler for memory optimization and parallelism management. In *PLDI*, 2010.

[48] Y. Zhang and F. Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, 2012.